Course Name: Cloud Computing

Project: Disjoint Set Union Implementation using Hadoop and PySpark

Instructor: Professor Maria Striki

Submitted by: **Group #3**

Aniqa Rahim - afr64@scarletmail.rutgers.edu
Shaan Kalola - spk103@scarletmail.rutgers.edu
Shivani Sunil - ss3013@scarletmail.rutgers.edu

Introduction

For this project, we implement a disjoint set union data structure which keeps track of the connected components of the graph. We implemented the algorithm using Hadoop as well as PySpark. We then came up with a hypothesis intertwining several factors, and then used that to compare and contrast the results obtained.

Motivation

The motivation for this paper was to increase the DSU performance by using the MapReduce paradigm. With Big Data being an ever growing field, DSU would benefit greatly with increased performance and would open a lot of doors for distributed implementation on the cloud.

Goal & Hypothesis

The goal was to implement DSU using Hadoop, we had to work around according to the number of cores in the system (2 sockets, 4 CPUs each = 8 cores). When it comes to Big Data, a larger number of cores would result in a greater performance efficiency. We looked into implementing our program on the cloud using an AWS EMR instance as well as Microsoft Azure HDinsight, but due to the large financial costs required to set this up we decided to use a shared Linux VM and installed Hadoop on there. We implemented DSU using PySpark as well, since we had the time to do so. We encountered no challenges with PySpark. However, when we tried to implement parallelization for faster performance, we had an error creating a DSU RDD due to virtual hardware restrictions. We did some research on the error, but could not find the solution.

Hypothesis - We expected the MapReduce to be faster on larger data sets as MapReduce executes in parallel, creates large trees fast, and has a tendency to combine large trees with smaller ones. If we combine a big tree with a smaller tree when we combine nodes from the smaller tree with new nodes we have to go through path compression which is O(log(n)). But the way MapReduce works is it will build a large tree for the first keys it accesses, and later keys

will be very small and take near constant time. We expected the MapReduce to be slower on smaller data sets.

Resources Used and Related Work:

- Hadoop
- PySpark
- MapReduce
- VM — MobaXterm
    - Linux OS
- MobaXterm Text Editor
- Multiple custom files of undirected graph data
    - 1,000 to 100,000,000 edges

Hadoop Implementation

Data Generation/Input File:

The input was created using a Java script for Hadoop as well as a Python script for PySpark. The file was randomly generated using Math.random functions and the data ranged from 10^3 to 10^8 edges.

Procedure:

- To implement DSU in hadoop we have to split our code into two main parts:
    - The first part being the DSU algorithm
    - The second part being the MapReduce
- We used the Hadoop dependencies as well as the makefile from lab 1 (modified) to implement this code.
- The time to implement the DSU operation with MapReduce was also recorded for each of the 6 files generated and a trend analysis of the data was done.
- The number of mappers was set using the input split method in the config file as well as in the code (included in the zip file).
- The number of reducers was set using the .JobSetMethod and it was equal to the number of cores in the Linux OS VM.
- There is no way to pass custom Data Structure from Configuration/Job Object to

Mapper/Reducer

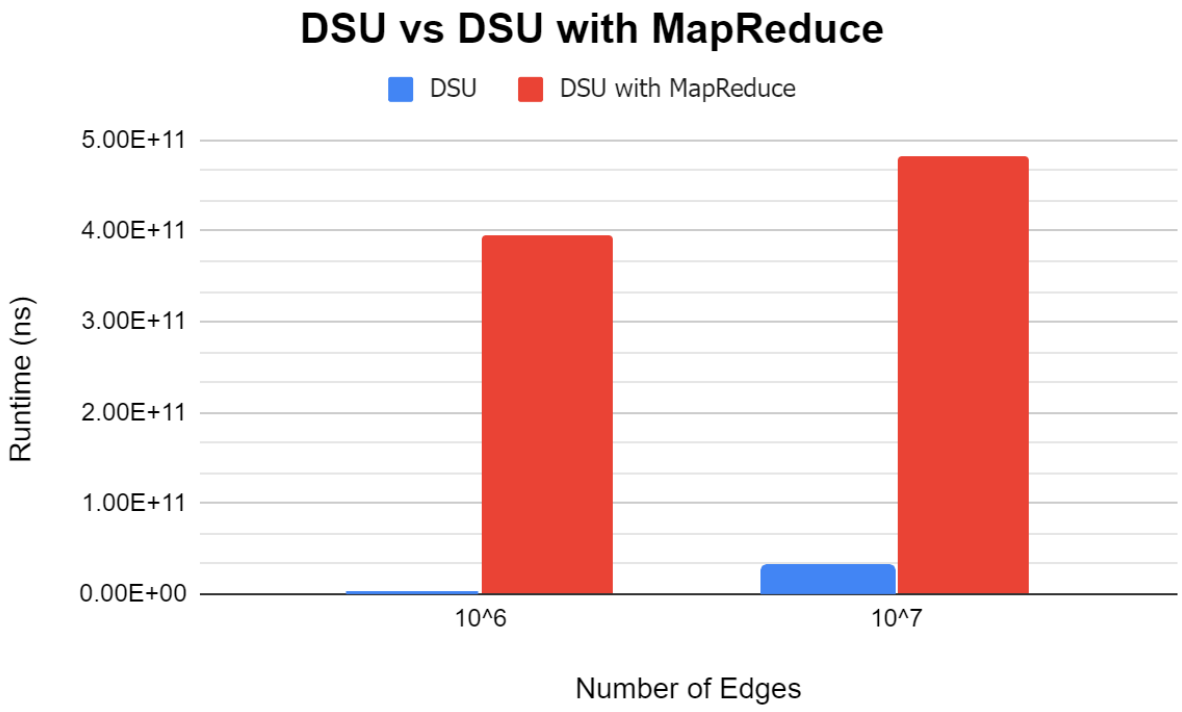
- You could convert your data to a byte array then write it as a string with a token, parse the string, and reconvert to a byte stream, and then convert to the Data Structure. Note: This method will not scale for large data.

Results:

The graph below depicts the performance for each of the files. We implemented it only for 2 different file sizes due to virtual hardware restrictions. The graph continues to prove our initial hypothesis we talk about in the analysis section.

Data:

Number of Edges	DSU Runtime (ns)	Number of Edges	DSU with MapReduce Runtime (ns)
10^7	3.04E+10	10^7	4.82E+11
10^6	2.09E+09	10^6	3.96E+11



The source code and results: **SEE ZIP FILE**

Analysis:

In this case, our hypothesis holds true DSU works faster with the MapReduce paradigm when it comes to larger data sets.

PySpark Implementation

Procedure:

- The input script written in Python was used to generate the input data for 6 runs.
- The DSU algorithm was implemented and then MapReduce with parallelization was implemented as well.
- The time.time() function was used to record the 6 runs with and without MapReduce and the graph was plotted.
- Attempted to make DSU a parallel RDD
    - Used one RDD to map edges (map u to list of v) then perform Union(u,v) on a parallel array that contains the DSU information

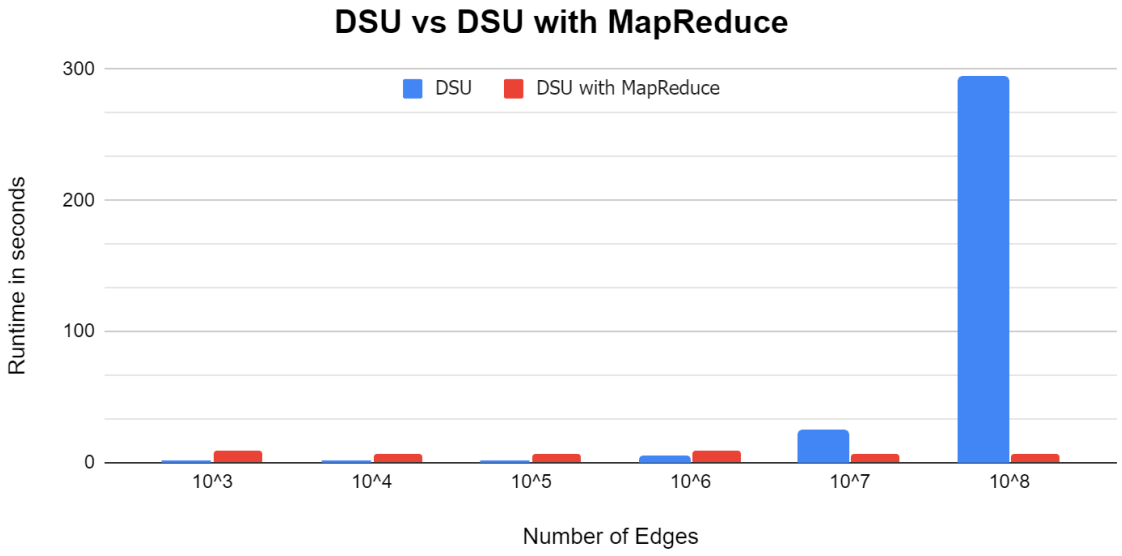
Results:

The graph below depicts the performance for each of the files. We implemented it for 6 different file sizes. The graph continues to prove our initial hypothesis we talk about in the analysis section.

Data:

Number of Edges	DSU Runtime (s)
10^8	293.744271
10^7	23.79186893
10^6	3.974918842
10^5	0.4120869637
10^4	0.06601977348
10^3	0.006296873093

Number of Edges	DSU with MapReduce Runtime (s)
10^8	6.96427989
10^7	7.223848343
10^6	9.26619792
10^5	6.890718937
10^4	7.215362072
10^3	9.08889699



Source Code: SEE ZIP FILE

Analysis:

In this case, our hypothesis holds true DSU works faster with the MapReduce paradigm with smaller datasets in PySpark. Additionally, it can be extrapolated that PySpark would take about 50 minutes to compute for 10^9 edges.

Conclusion and Class Related Work

The project was centered around implementing an algorithm using the two biggest Big Data frameworks taught in this course, and using MapReduce to the best of our ability to prioritize computational efficiency. The main advantage of Big Data is efficient processing of large amounts of data that incur minimal cost. DSU is a data-heavy algorithm and combining it with data mining tools was a wise decision. We demonstrated through our project the difference in performance caused by just implementing MapReduce on a VM with 8 cores, and if this extrapolated to a bigger setting and done by hosting it on AWS or Microsoft Azure, the results would be impactful to say the least. Computational efficiency has always been at the forefront of programming and software development, with this project we prove that existing algorithms can be implemented in a better way by using the required frameworks. Our hypothesis holds true even after the data was collected; although we did face some challenges with Hadoop as the data

generated was too much to be handled by the number of cores we had available to us. Ultimately, using the common case of computing, we used the existing VM instead of increasing the cores as that would result in a cost surge and we also wanted to experiment parallelization by setting the mappers and reducers to the number of cores (8 in the case of Hadoop, as it it a heavy framework; and 4 cores in the case of PySpark).

The main conclusion of this project was that DSU performance significantly improves when implemented along with MapReduce, a combiner, would make it even faster but the difference would be very menial. However we also notice that MapReduce does not work well with large datasets in PySpark which is where Hadoop, the big data framework, comes in as the runtime was in nanoseconds when compared to seconds in PySpark. Mapping and reducing is an expensive operation and we learned that it does not make sense when the dataset to be used is smaller. Hence, the hypothesis holds true for data above a certain threshold. The threshold can further be determined in the future when it is hosted on cloud.

The takeaways and learning experiences ranged from how to integrate what is learned in a class setting in a completely different type of algorithm and how computation can always be made efficient. It comes down to the decision of cost vs. efficiency and the former is often the controlled variable and latter is tested. Another aspect we took away from this lab is the importance of makefiles when it comes to an industry setting or with code that involves many components; a makefile makes working with Hadoop a lot easier and faster.

MapReduce has been one of the main topics of Cloud Computing and how it can be used in various settings whether it is WordCount or in PageRank; combining data structures with cloud computing was something that we have implemented since lab 1 and in this project we just beyond and selected a different category of an algorithm. This enables us to test the skills we acquired through the labs and papers to produce accurate results based on the proposal we had originally submitted.

Individual Contribution:

Aniqa Rahim	● Created the tables and graphs for the presentation ● Worked on the dsu_MapReduce Hadoop code ● Operated the VM (MobaXterm) ● Worked on the project report ● Worked on the presentation and presented
Shivani Sunil	● Worked on the DSU/MapReduce PySpark code ● Worked on the dsu_MapReduce Hadoop code ● Worked on the project report ● Worked on the presentation and presented
Shaan Kalola	● Collected the output data ● Worked on the dsu_MapReduce Hadoop code ● Wrote the scripts to generate the input data ● Worked on the dsu_MapReduce PySpark code ● Worked on the project report ● Worked on the presentation and presented